# オペレーティングシステム特論　課題

講義で「OS は特権モードで動いている」ことを説明した。この課題では特権モードでしか実行できない命令（特権命令）を一般プログラムと OS 内で実行することで、実際に OS が特権モードで動いていることを確認する。

対象とする特権命令は rdmsr 命令である。この命令は Model Specific Register (MSR) の値を読み出す。MSR とは CPU の設定を行うレジスタであり、MSR を読み書きすることで CPU が特定の機能をサポートしているかを知ったり特定の機能のオンオフを切り替えたりできる。計算に使う通常のレジスタ（General Purpose Register）は同じ命令セットを実装する CPU であれば共通に存在するのに対し、各 MSR は CPU モデルによって存在したりしなかったりするため Model-Specific である。

C コンパイラのインストール

linux_install.pdf に従ってインストールした Debian に ssh クライアントを用いてログインし、以下のコマンドを実行して本課題に必要な C コンパイラをインストールする。

```
sudo apt-get install gcc
```

rdmsr の一般プログラム内での実行

C 言語から特定の機械語命令を直接呼び出すにはコンパイラのインラインアセンブラという機能を使う。これは C のコード内にアセンブリ言語のプログラムを直接書ける機能である。rdmsr 命令を呼び出すコード例を以下に示す。

```
#include <stdio.h>

int main() {
    asm volatile(“rdmsr;”);
    printf(“hello¥n”);
    return 0;
}
```

これを rdmsr_user.c として保存、コンパイルし、rdmsr 命令が実際にプログラム内に含まれていることを確認する。

```
gcc rdmsr_user.c -o rdmsr_user
objdump -d rdmsr_user | grep rdmsr
rdmsr_user:     ファイル形式 elf64-x86-64
      1139:      0f 32          rdmsr
```

objdump は機械語のオブジェクトプログラムを逆アセンブルし人間が分かる形式で表示するツールである。この例では先頭から 0x1139 バイト目に rdmsr 命令が含まれている。

できたプログラムを実行すると、以下のように強制終了する。hello が表示されていないことから、printf(“hello¥n”) に到達する前に強制終了していることが分かる。

```
./rdmsr_user
Segmentation fault
```

なお講義で扱ったように root 権限と特権モードは独立の概念であり、root 権限であっても非特権モードでは特権命令は実行できない。実際に sudo をつけて root 権限で実行しても同じ結果になる。

```
sudo ./rdmsr_user
Segmentation fault
```

最近のバージョンの Linux では、segmentation fault 時に詳細をカーネルメッセージとして出力してくれる。カーネルメッセージの一覧は dmesg コマンドで取得可能である。具体的に、rdmsr_user に関するカーネルメッセージは以下のように取り出せる。dmesg コマンドの実行には root 権限が必要であることに注意。

```
sudo dmesg | grep rdmsr_user
[sudo] user のパスワード:
……
```

**問 1:** 難易度 0/3

rdmsr_user を実行したのち、カーネルメッセージの rdmsr_user に関する部分を表示せよ（即ち上記のコマンドを実行せよ）。表示されたメッセージの内容（上で …… とした部分）を提出せよ。

**問 2:** 難易度 1/3

rdmsr_user の実行において printf に到達する前に強制終了されカーネルメッセージが表示された。つまりこれはプログラムの処理順序が rdmsr 命令と printf （を呼び出す命令）の間で強制的に変更され、OS に処理が移ったことを意味する。これは「どういう理由で何が起こり、その結果 OS 内の何に処理が移った」と考えられるか答えよ。

rdmsr の OS 内での実行

次に rdmsr 命令を OS 内で実行したい。OS 本体のコードは膨大でコンパイルに長い時間がかかるため、ここではカーネルモジュール内で rdmsr を呼び出す。カーネルモジュールとは、OS のカーネルとは別にコンパイルしたオブジェクトプログラムをカーネルに動的リンクする機構である。カーネルモジュールはカーネルとは独立してコンパイルされるが、カーネルに動的リンクされると OS の一部として特権モードで動作し、カーネル内の全ての公開関数やグローバル変数を参照できる。

**問 3:** 難易度 2/3

Linux はモノリシックカーネルであると講義で述べたが、単純なモノリシックではなくカーネルモジュールによって機能を後付けできる。単純なモノリシックカーネルと比較して、カーネルモジュールを用いることの利点を 1 つあげ、具体的にその利点が役に立つユースケースを用いて説明せよ。

まずカーネルモジュールの作成の練習をする。ダウンロードした rdmsr_kernel.c と Makefile を同一のディレクトリ（os_homework とする）内に置き、下記の手順を実行せよ。ファイル名を rdmsr_kernel.c 以外にすると失敗するので注意すること。

1. 以下のコマンドを実行し、カーネルモジュールのコンパイルに必要なパッケージをインストールする。uname と-r の間に半角スペースがあり、`uname と の間や – と ` の間にはスペースがないことに注意せよ。 ` は「バッククォート」であり日本語キーボードでは Shift + @ で入力できる。

```
sudo apt-get install build-essential linux-headers-`uname -r`
```

2. rdmsr_kernel.c と Makefile を格納したディレクトリで make を実行する。

```
cd os_homework
make
```

…

os_homework 内に rdmsr_kernel.ko ができていれば成功であり、これがカーネルモジュールのオブジェクトプログラムである。カーネルモジュールは以下の手順でカーネルに組み込んだりカーネルから取り外したりできる。これらの手順は必ずセットで行い、モジュールを取り外す前に同じモジュールを再度組み込まないこと。

本モジュールは start 関数の中で printk 関数を呼び出している（printf ではない）。printk はカーネルメッセージを出力する関数であり、引数の頭に KERN_INFO をつけること以外 printf と同様に使用できる。KERN_INFO と文字列の間にカンマがないことに注意せよ。出力結果は dmesg コマンドで確認できる。なお出力結果の数字は OS 起動からの経過秒数であるので実行例と異なる場合がある。

モジュールをカーネルに組み込み

```
sudo insmod rdmsr_kernel.ko
```

カーネルメッセージを確認

```
sudo dmesg | grep rdmsr_test
[499.586779] rdmsr_test start
```

モジュールをカーネルから取り外し

```
sudo rmmod rdmsr_kernel.ko
```

次にカーネルモジュール内で rdmsr 命令を実行する。一般プログラム内では単に rdmsr 命令が実行できないことを確認したが、今回は得られる結果の正しさも確認したい。rdmsr 命令で読み出す MSR 番号は ecx レジスタで指定し、読み出し結果の上位 4 バイトが eax レジスタに、下位 4 バイトが edx レジスタに入る。例えば 0x10 番の MSR の内容が 0x123456789ABCDEF0 であるとき、ecx レジスタに 0x10 をセットし rdmsr 命令を実行すれば eax レジスタは 0x12345678 に、edx レジスタは 0x9ABCDEF0 になる。

**問 4：** 難易度 3/3

rdmsr_kenel.c の start 関数を変更し、0xC0000080 番の MSR を読み出しその結果を表示せよ。できたプログラムを提出せよ。

ヒント 1：インラインアセンブラを用いて rdmsr 命令を呼び出す前に ecx レジスタに 0xC0000080 をセットし、その後 eax レジスタと edx レジスタの結果を C 側に取り出して printk すればよい。インラインアセンブラの使い方については以下のサイトなどを参考にせよ。なお __asm__() と asm volatile() と asm() の違いは今回は気にせず、asm volatile() で統一せよ。

http://caspar.hazymoon.jp/OpenBSD/annex/gcc_inline_asm.html
https://msyksphinz.hatenablog.com/entry/2016/03/28/020000

ヒント 2：asm volatile() の第一引数は文字列であるが、C では複数の文字列を ¥ （またはバックスラッシュ）でつなぐことで一つの文字列として扱える。

```
printf("This is " ¥
       "recognized as " ¥
       "a single " ¥
       "string¥n"
       );
```

**問 5：** 難易度 1/3

問 4 で読み出した結果を提出せよ。なお “Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 4: Model-Specific Registers” によれば、0xC0000080 番の MSR の仕様は以下のようになっている。“Reserved” とは将来の使用のために予約されているという意味であり、“Reserved” なビット（ビット 1 からビット 7、ビット 9、ビット 12 からビット 63）は 0 であるはずである。

| | | | | |
|---|---|---|---|---|
| C000_0080H | | IA32_EFER | Extended Feature Enables | If ( CPUID.80000001H:EDX.[20] \|\| CPUID.80000001H:EDX.[29]) |
| | | 0 | SYSCALL Enable: IA32_EFER.SCE (R/W) Enables SYSCALL/SYSRET instructions in 64-bit mode. | |
| | | 7:1 | Reserved | |
| | | 8 | IA-32e Mode Enable: IA32_EFER.LME (R/W) Enables IA-32e mode operation. | |
| | | 9 | Reserved | |
| | | 10 | IA-32e Mode Active: IA32_EFER.LMA (R) Indicates IA-32e mode is active when set. | |
| | | 11 | Execute Disable Bit Enable: IA32_EFER.NXE (R/W) | |
| | | 63:12 | Reserved | |

参考情報

本課題では仮想マシンの中で特権命令を実行し無事に実行できるように見えたが、よく考えるとこれは問題である。例えば二つの仮想マシンが同じホスト OS（仮想マシンの下で動いている本当の OS）の上に存在するとき、一方の仮想マシンが自由に特権命令を実行できるならばもう一方の仮想マシンを任意に操作できる。さらに仮想マシンが特権命令を実行することでホスト OS をも任意に操作することができてしまう。

実際にはこんなことが起こらないように、仮想マシンが特権命令を実行すると例外が発

生しホスト OS に処理が移り特権命令がエミュレートされる。これにより仮想マシンから見ると特権命令が仕様通り実行されたように見えるのである。この事について詳しくは仮想化についての講義で説明する。